

When every FLOPS counts: GPU Sharing Strategies at CERN

GOTO **COPENHAGEN**

Craft software with confidence in 2026

QUESTIONS?

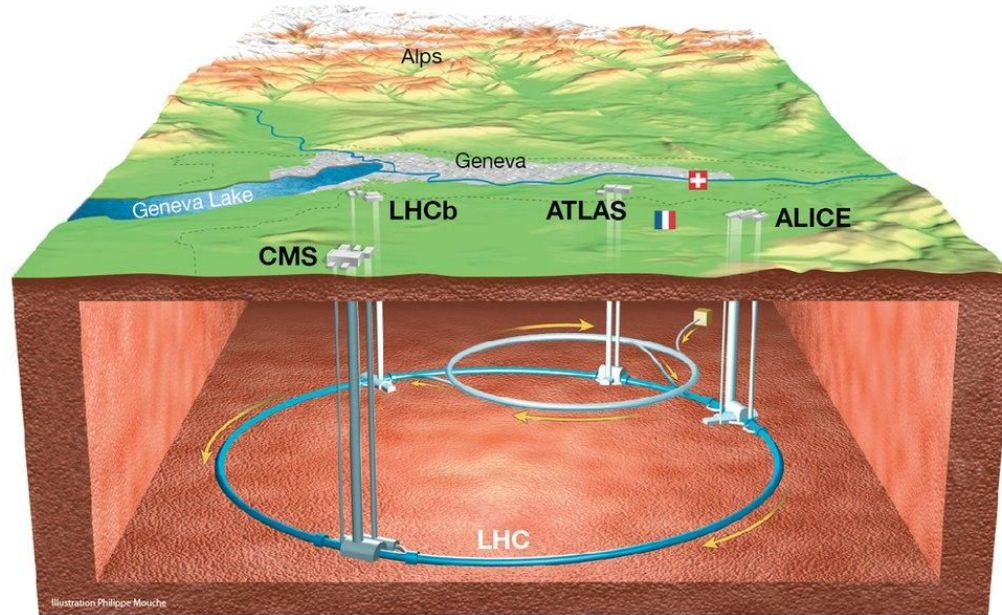
ASK ME IN THE APP



Diana Gaponcic - diana.gaponcic@cern.ch
Computing Engineer with the Kubernetes team @ CERN



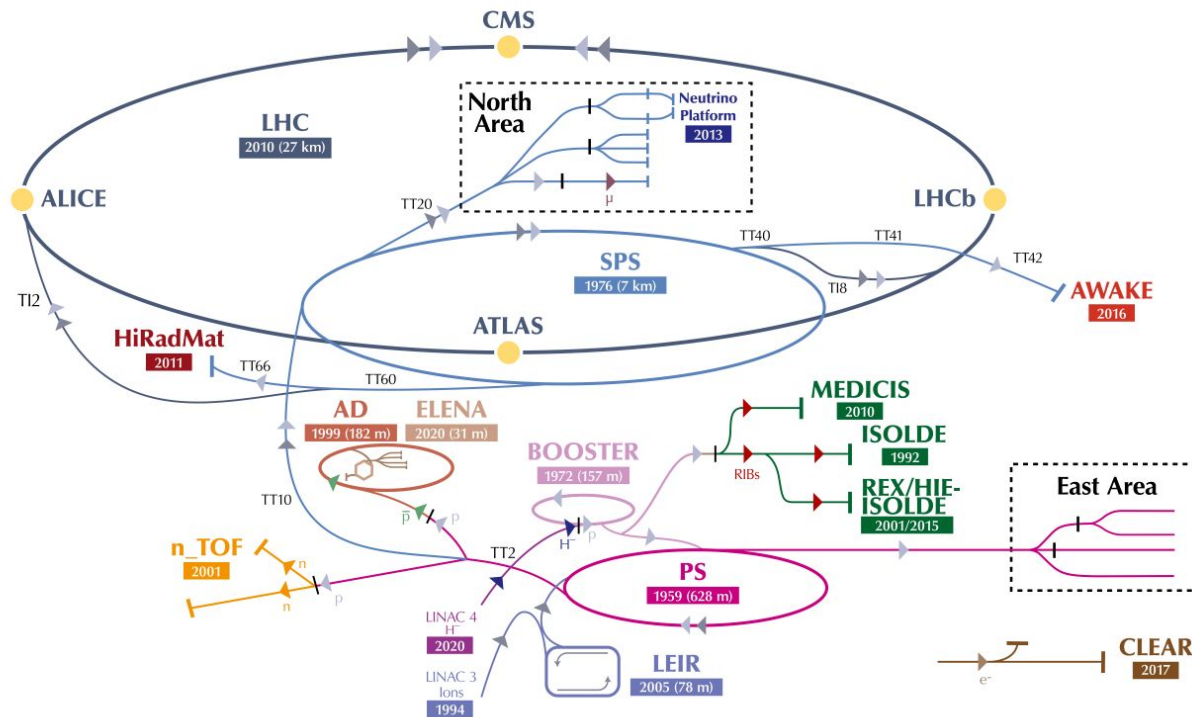
What is CERN



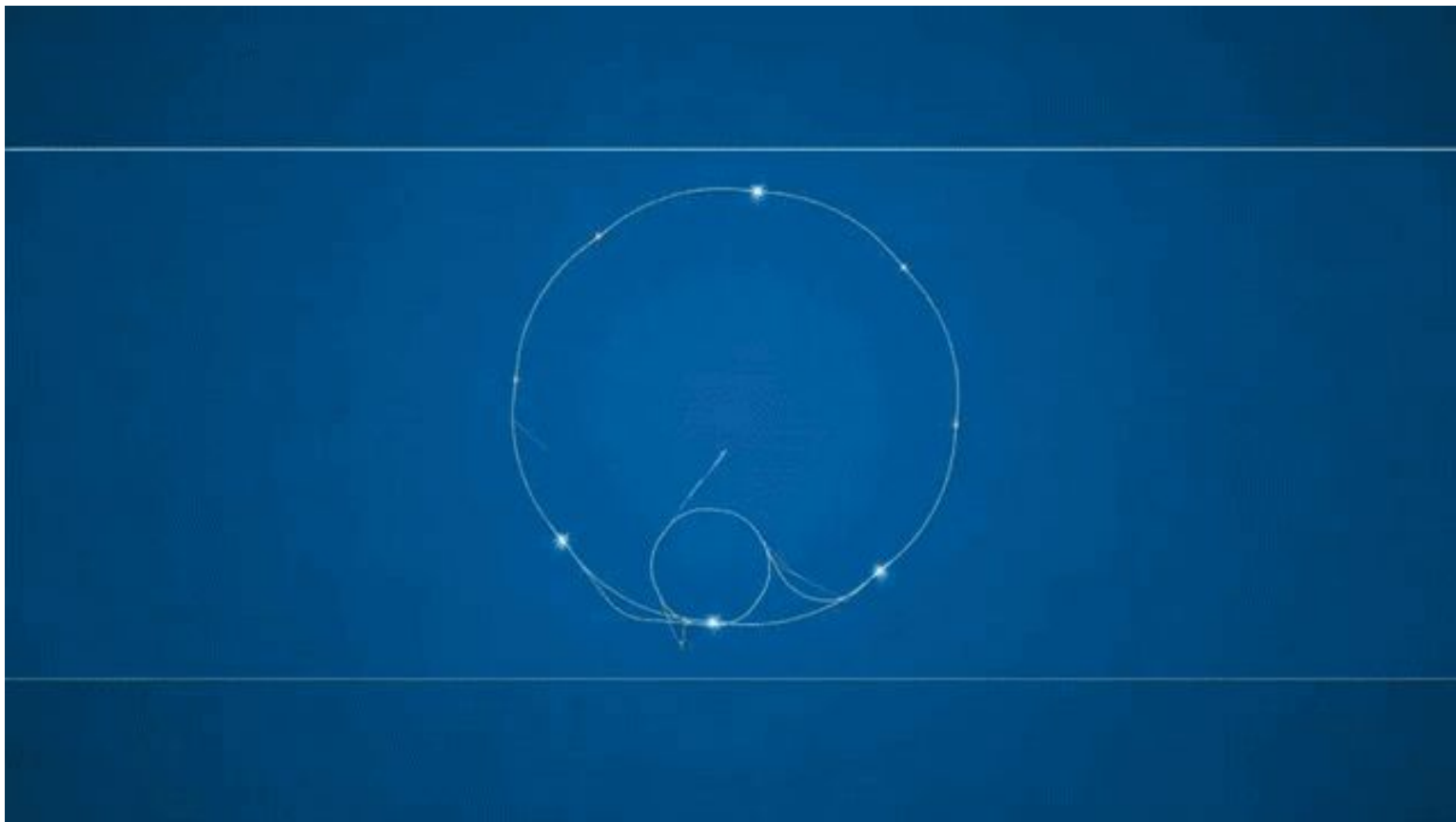
Awesome CERN overview video: <https://www.youtube.com/watch?v=pQhbhpU9Wrg>

The CERN accelerator complex

Complexe des accélérateurs du CERN



Source: <https://home.cern/science/accelerators/the-accelerator-complex/>



Awesome CERN overview video: <https://www.youtube.com/watch?v=pQhbhpU9Wrg>

Why CERN needs GPUs?

[Inspire HEP](#)

Search: GPU

Date of paper



Search: AI

Date of paper



Search: ML

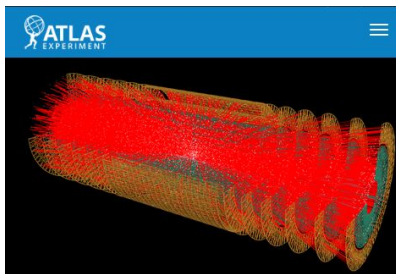
Date of paper



* INSPIRE is a trusted community hub that helps researchers to share and find accurate scholarly information in high energy physics.

Usage of GPUs in ALICE Online and Offline processing during LHC Run 3

[Source](#)



[updates](#) > [news](#) >
Are you up for the TrackML challenge?

News

Tags:
machine learning

Are you up for the TrackML challenge?

Can machine learning assist high-energy physics in discovering and characterising new particles?

[Source](#)

[Voir en français](#)

Allen initiative – supported by CERN openlab – key to LHCb trigger upgrade

From 2021, the first stage of LHCb's high-level trigger will be run using GPUs. Investigations into its feasibility were supported by CERN openlab

[Source](#)



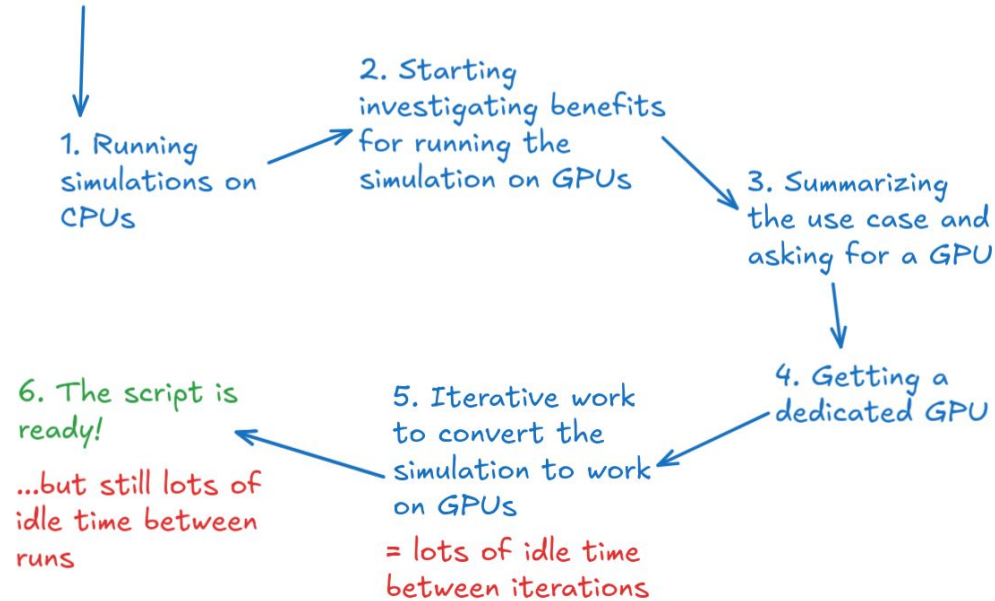
[Source](#)

LHC experiments are stepping up their data processing game

While data processing demand is rocketing for LHC's Run 3, the four large experiments are increasing their use of GPUs to improve their computing infrastructure

[Source](#)

What often happens...

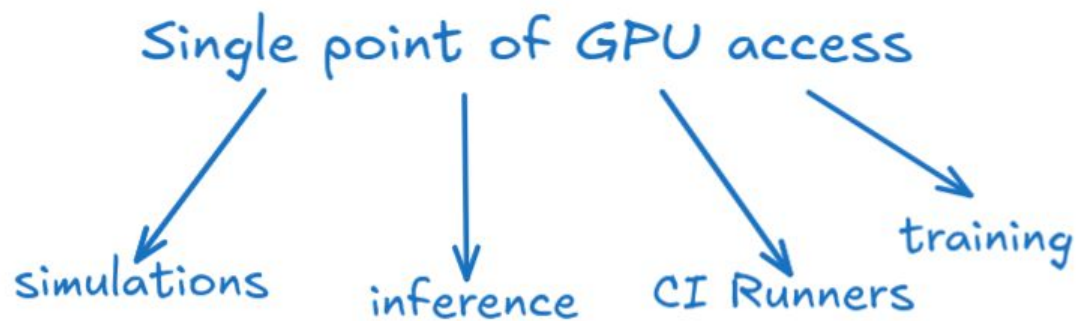


What can we do?

First, centralize your GPU metrics:

1. Get access to the GPU metrics from different teams/service
2. Normalize the data
3. Build a single dashboard overviewing all the various GPU usage utilization patterns
4. Compare

Then centralize your GPUs:



Alright, at least now all the GPUs are in-use all the time!

Alright, at least now all the GPUs are in-use all the time!
...Or are they?

Alright, at least now all the GPUs are in-use all the time!
...Or are they?

Even if the GPU are always in-use, many use cases
will not fully utilize the GPUs
=> A lot of wasted idle resources

Alright, at least now all the GPUs are in-use all the time!
...Or are they?

Even if the GPU are always in-use, many use cases
will not fully utilize the GPUs
=> A lot of wasted idle resources

1. If GPU allocated but no utilization
=> Cull the user

Alright, at least now all the GPUs are in-use all the time!
...Or are they?

Even if the GPU are always in-use, many use cases
will not fully utilize the GPUs
=> A lot of wasted idle resources

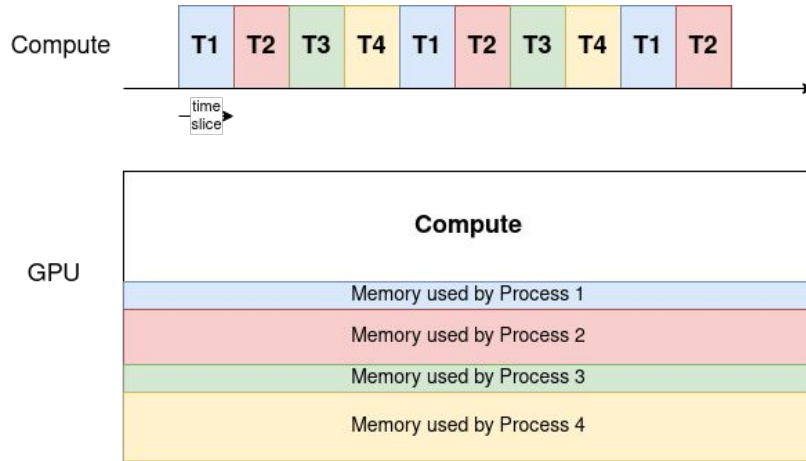
1. If GPU allocated but no utilization
=> Cull the user

2. Otherwise, this is where GPU Sharing comes into play!

GPU Sharing Mechanisms: Time-slicing

Time-slicing

- The scheduler gives an equal share of time to all GPU processes and alternates them in a round-robin fashion.
- The memory is shared between the processes
- The compute resources are assigned to one process at a time



Disclaimer 1: This presentation focuses a lot
on Kubernetes

Disclaimer 1: This presentation focuses a lot
on Kubernetes

Disclaimer 2: To access GPUs on Kubernetes
we've been using the k8s-device-plugin

The [NVIDIA device plugin](#) for Kubernetes is a Daemonset that allows you to automatically:

- Expose the number of GPUs on each nodes of your cluster
- Keep track of the health of your GPUs
- Run GPU enabled containers in your Kubernetes cluster.
- Expose the number of shared GPUs

Disclaimer 1: This presentation focuses a lot on Kubernetes

Disclaimer 2: To access GPUs on Kubernetes we've been using the k8s-device-plugin

The [NVIDIA device plugin](#) for Kubernetes is a Daemonset that allows you to automatically:

- Expose the number of GPUs on each nodes of your cluster
- Keep track of the health of your GPUs
- Run GPU enabled containers in your Kubernetes cluster.
- Expose the number of shared GPUs **-> This is called system-mediated sharing**

How to enable and use Time-slicing

Allocatable:
...
nvidia.com/gpu: 1

values.yaml in NVIDIA gpu operator Helm chart

```
...  
devicePlugin:  
  config:  
    name: nvidia-time-slicing-config
```

\$ cat nvidia-time-slicing-config.yaml

```
apiVersion: v1  
kind: ConfigMap  
metadata:  
  name: nvidia-time-slicing-config  
  namespace: kube-system  
data:  
  slice-4: |-  
    version: v1  
    sharing:  
      timeSlicing:  
        renameByDefault: true  
        failRequestsGreaterThanOne: true  
        resources:  
        - name: nvidia.com/gpu  
          replicas: 4
```

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: tf-gpu  
spec:  
  containers:  
    - name: tf  
      image:  
        tensorflow/tensorflow:latest-gpu  
      command: ["sleep", "inf"]  
      resources:  
        limits:  
          [nvidia.com/gpu.shared: 1]
```

Allocatable:
...
nvidia.com/gpu: 0
[nvidia.com/gpu.shared: 4]

```
kubectl label node <node-name> nvidia.com/device-plugin.config=slice-4
```

Time-Slicing

Advantages

Works on a wide range of NVIDIA architectures

An easy way to set up GPU concurrency

An unlimited number of partitions

Disadvantages

No memory isolation

No ability to set priorities

Inappropriate for latency-sensitive applications (ex: desktop rendering for CAD workloads)

How will time-slicing perform on a (somewhat) real example?

Meet the actors: Ms. Cold brew

been "working on it" since yesterday



Badly coded simulation job:

- Low average GPU usage (CPU dependant workload)
- Needs 10 GiB VRAM (8 + 2 dynamic)
- Long running process

Meet the actors: Mr. Espresso shot

hold...hold...hold...go!

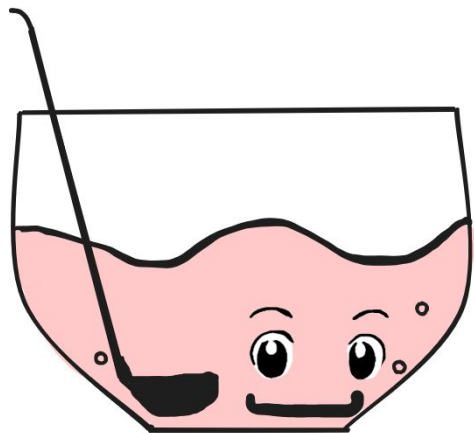


An inference service which is occasionally triggered by outside events:

- Spiky and unpredictable execution
- Mostly sits idle
- Saturates the GPU cores
- Max 10 GiB VRAM (2 + 8 dynamic)

Meet the actors: Mr. Mystery punch bowl

could be anything, might overflow



Never know what to expect from a notebook user:

- Potential memory leaks
- Poorly considered batch size
- GPU memory locked by an idle notebook

* All use cases were run on a CERN Kubernetes cluster with 1 NVIDIA A100 40GB GPU

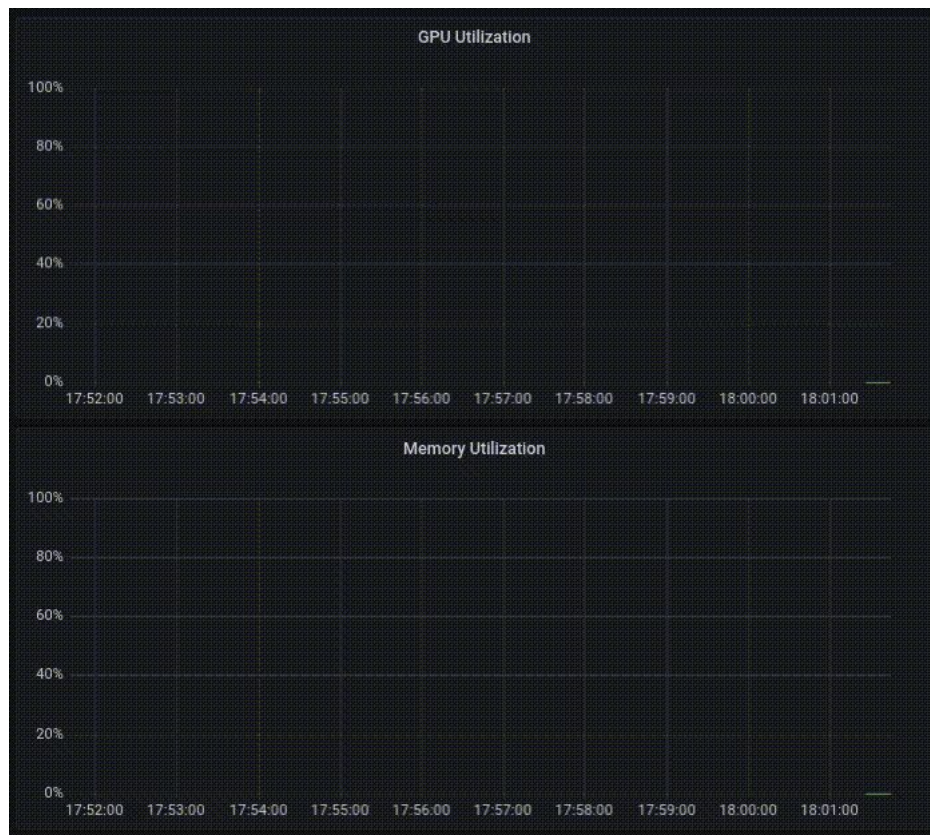
Onboard **Only** Actor 1 = Dedicated GPU

Badly coded simulation job:

- Low average GPU usage (CPU dependant workload)
- Needs 10 GiB VRAM (8 + 2 dynamic)
- Long running process



- GPU underutilized
- Steady memory utilization ~ 20%



Actor 1



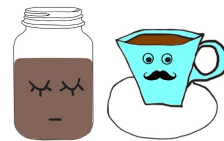
- GPU underutilized
- Steady memory utilization
~ 20%

Actors 1 & 2

* Time-Slicing GPU Sharing



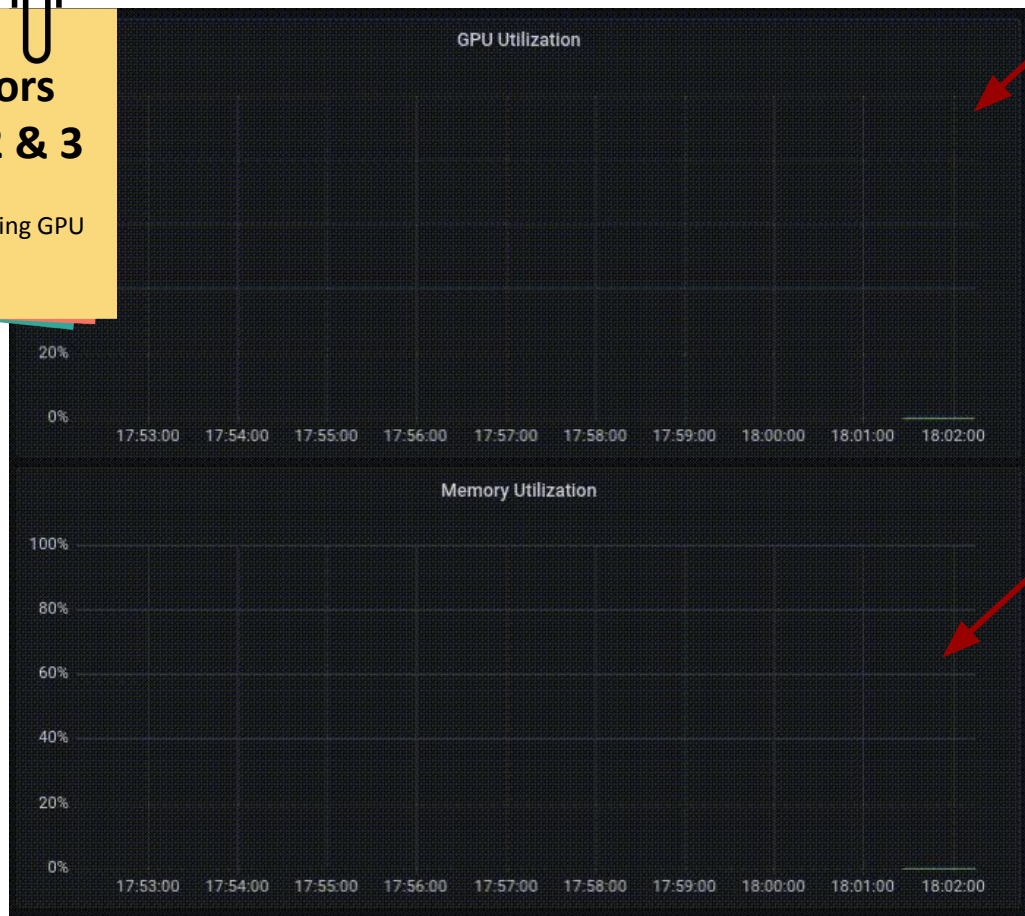
- Improved GPU utilization
- Better memory consumption (~ 50 %)



Actors

1 & 2 & 3

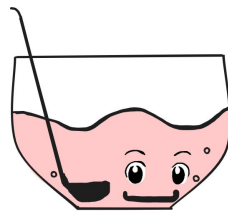
* Time-Slicing GPU Sharing



GPU utilization 100%

... Perfect, right?

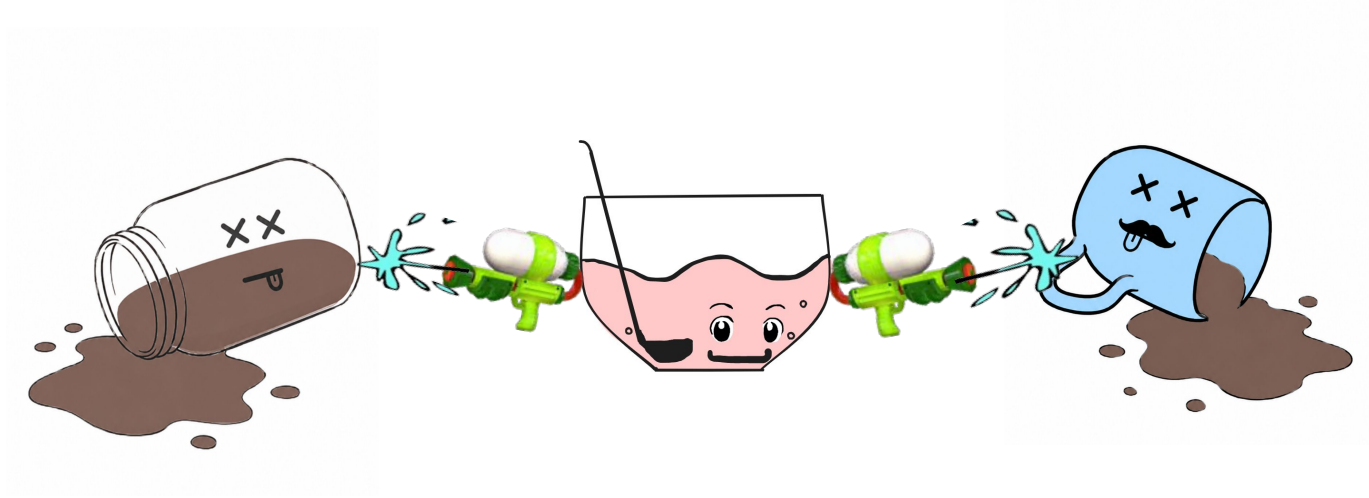
Not really



Use case 3 used all the memory, and **starved** the other 2 processes, and continued running.



No Amount of coffee will save you from the treacherous punch



Ms. Cold Brew
cause of death: **OOM**

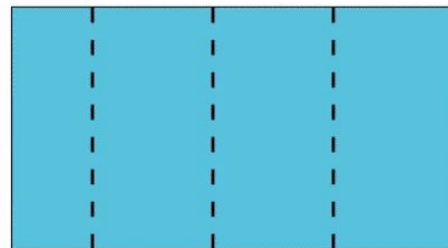
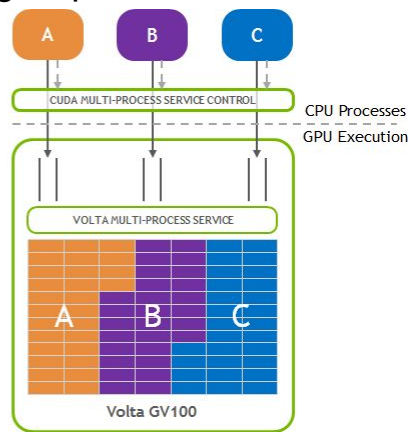
Mr. Espresso shot
cause of death: **OOM**

GPU Sharing Mechanisms: Multi-Process Service

Multi-Process Service

Multi-Process Service (MPS) is a runtime architecture to enable cooperative multi-process CUDA applications.

- MPS control daemon manages the amount of memory allowed per client
- MPS control daemon manages the amount of compute capacity that can be consumed by a client.
- No context switching required



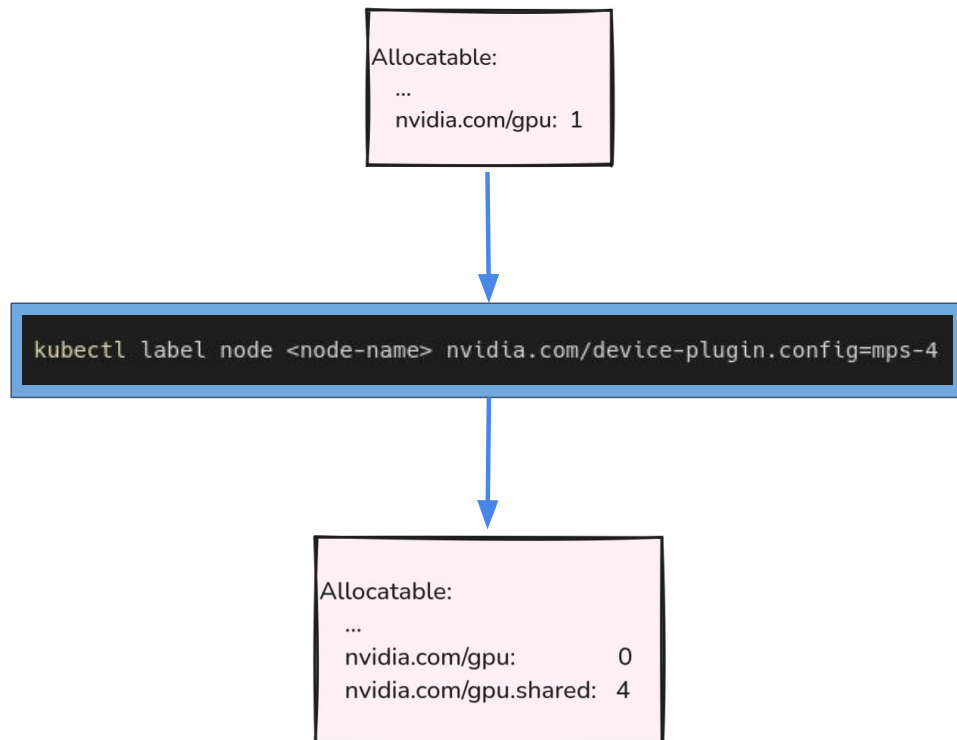
Source: <https://docs.nvidia.com/deploy/mps/index.html>

How to currently enable MPS

```
# values.yaml in NVIDIA gpu operator  
Helm chart
```

```
...  
devicePlugin:  
  config:  
    name: nvidia-config
```

```
# $ cat nvidia-config.yaml  
apiVersion: v1  
kind: ConfigMap  
metadata:  
  name: nvidia-config  
  namespace: kube-system  
data:  
  mps-4: |-  
    version: v1  
    sharing:  
      mps:  
        resources:  
          - name: nvidia.com/gpu  
            replicas: 4
```



Multi-Process Service

Advantages

No context switching required

An easy way to set up GPU concurrency

Works on a wide range of GPU architectures

Disadvantages

Memory isolation limitations

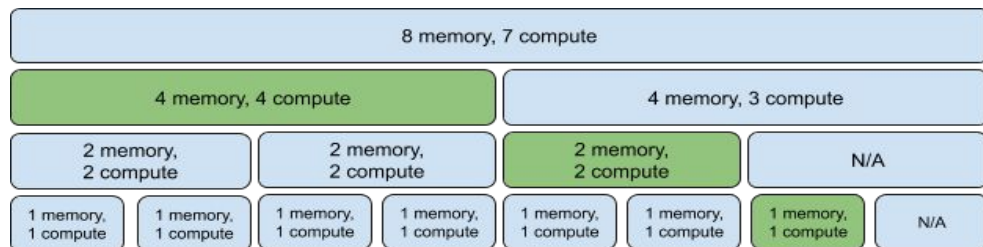
Hard to monitor

Resources used to manage the coordination and the resource usage of the processes

GPU Sharing Mechanisms: Multi Instance GPU

Multi Instance GPU

Multi Instance GPU (MIG) can partition the GPU into up to seven instances, each fully isolated with its own high-bandwidth memory, cache, and compute cores.



NVIDIA A100 GPU can be partitioned into up to 7 MIG devices

http://www.pattersonconsultingtn.com/blog/introduction_to_nvidia_mig.html



How to currently enable MIG

```
# values.yaml in NVIDIA gpu operator  
Helm chart
```

```
...  
mig:  
  strategy: mixed  
  migManager:  
    config:  
      name: nvidia-mig-config
```

```
apiVersion: v1  
kind: ConfigMap  
metadata:  
  name: nvidia-mig-config  
data:  
  config.yaml: |  
    version: v1  
    mig-configs:  
      # A100-40GB  
      3g.20gb-2x2g.10gb:  
        - devices: all  
          mig-enabled: true  
          mig-devices:  
            "2g.10gb": 2  
            "3g.20gb": 1
```

Allocatable:
...
nvidia.com/gpu: 1

kubectl label nodes <node-name> nvidia.com/mig.config=3g.20gb-2x2g.10gb

Allocatable:
...
nvidia.com/gpu: 0
nvidia.com/mig-2g.10gb: 2
nvidia.com/mig-3g.20gb: 1

Multi Instance GPU

Advantages

Hardware isolation allows processes to run securely in parallel and not influence each other

Monitoring and telemetry data available at partition level

Allows partitioning based on use cases, making the solution flexible

Disadvantages

Only available for Ampere, Hopper, and Blackwell architecture

* Potential loss of available memory depending on chosen profile layout

* Not a risk if the partitioning layout is chosen in an informed way after careful consideration.

-> Actor 1
-> Actor 2
-> Actor 3

* MIG GPU
Sharing



Every process:

- Is isolated
- Saturates own resources
- Cannot influence other processes

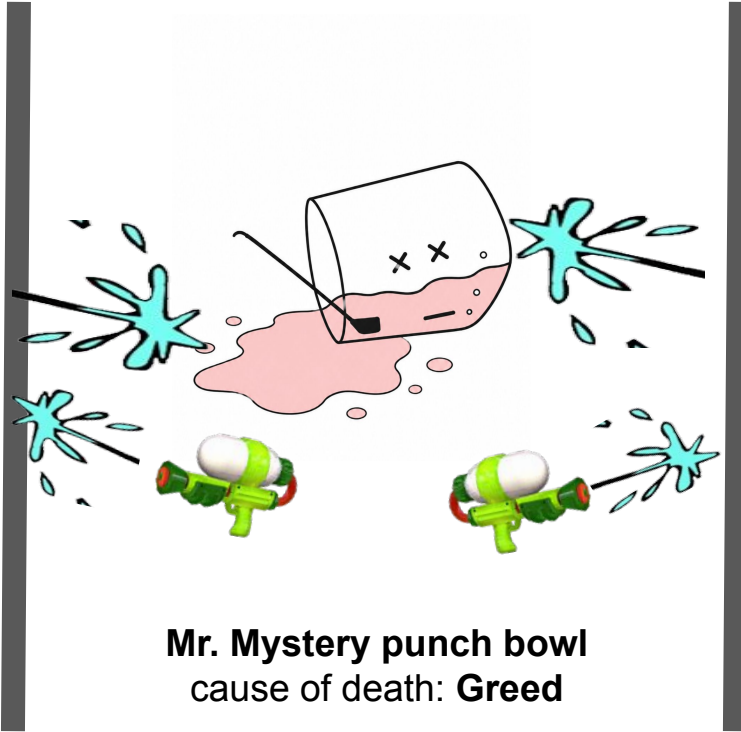
... Perfect, right?

Yes.

Use case 3 **starved itself**,
use cases 1 & 2 continued
running without issues!

Mr. Mystery punch bowl is isolated and cannot cause problems to others

Huh, now I have
enough time to
brew

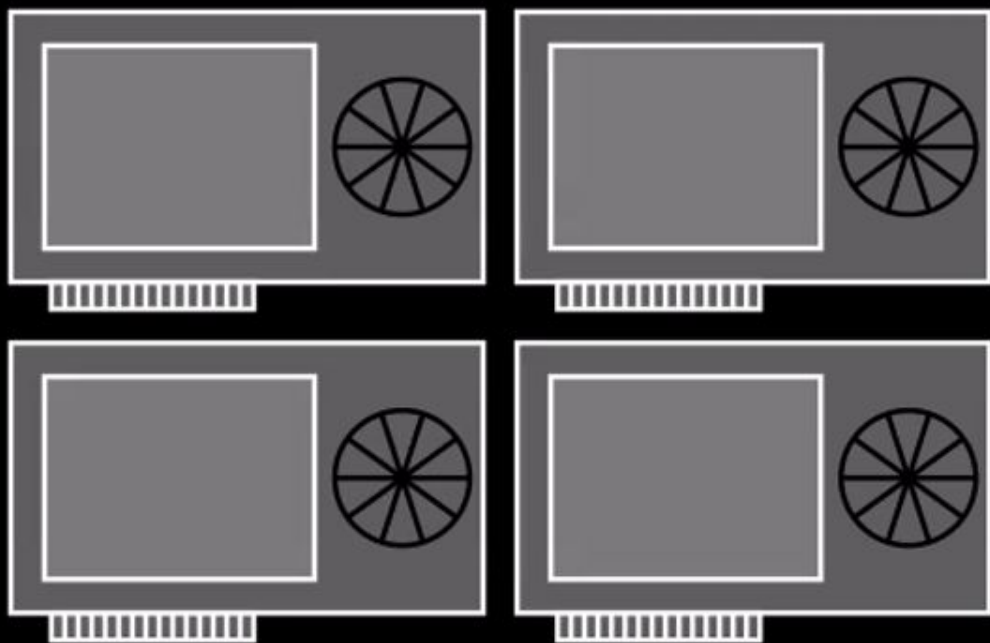


Mr. Mystery punch bowl
cause of death: **Greed**

Alive!
Hooray!



Let's see how the full system performs with
GPU sharing



Clear pain point:
Manual reconfiguration

Another clear pain point:
Some configuration options affect the **full node**

We have nodes with 2, 4, 8 GPUs, and we want to manage them
individually

Another clear pain point:

No possibility to express more complex requirements

You can't request multiple GPU types in order of priority.

If you request an H100, and it's not available, you might want to go for an A100 instead, but this is not something easily achievable.

Introducing DRA

Dynamic Resource Allocation (DRA) is an API for requesting and sharing resources between pods and containers inside a pod. It is a generalization of the persistent volumes API for generic resources. Typically those resources are devices like GPUs.

FEATURE STATE: **Kubernetes v1.34 [stable]** (enabled by default)

DRA is just an API, it's up to different vendors providing the hardware to implement the drivers to manage their specific devices.

If you want to find more on how to develop a driver, best resource:

[DRAcon: Demystifying Dynamic Resource Allocation - from Myths to Facts](#)
[- Kevin Klues & Patrick Ohly](#)

DRA is just an API, it's up to different vendors providing the hardware to implement the drivers to manage their specific devices.

If you want to find more on how to develop a driver, best resource:

[DRAcon: Demystifying Dynamic Resource Allocation - from Myths to Facts - Kevin Klues & Patrick Ohly](#)

*In March 2026, at **KubeCon Europe** in Amsterdam, two major vendors made coordinated moves toward open hardware orchestration. **NVIDIA** donated their **Dynamic Resource Allocation driver for GPUs** to the **Kubernetes** community. **Google** donated their DRA driver for Tensor Processing Units (TPUs). Source:*

<https://medium.com/@christian.dussol/from-counting-gpus-to-reasoning-about-them-how-kubernetes-dra-powers-ai-native-infrastructure-369cc75a667e>

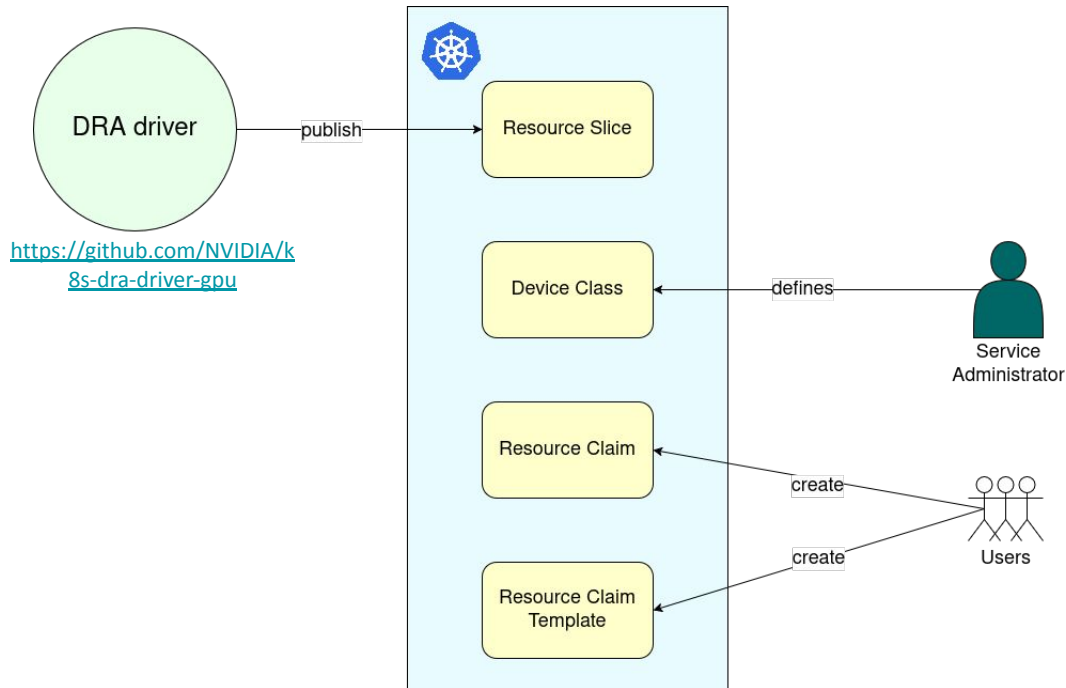
DRA is just an API, it's up to different vendors providing the hardware to implement the drivers to manage their specific devices.

NVIDIA: <https://github.com/kubernetes-sigs/dra-driver-nvidia-gpu>

Intel: <https://github.com/intel/intel-resource-drivers-for-kubernetes>

AMD: <https://github.com/ROCm/k8s-gpu-dra-driver>

Google TPUs: <https://github.com/kubernetes-sigs/dra-driver-google-tpu>



- **ResourceClaim:** Describes a request for access to attached resources, such as devices, in the cluster. ResourceClaims provide Pods with access to a specific resource. ResourceClaims can be created by workload operators or generated by Kubernetes based on a ResourceClaimTemplate.
- **ResourceClaimTemplate:** Defines a template that Kubernetes uses to create per-Pod ResourceClaims for a workload. ResourceClaimTemplates provide Pods with access to separate, similar resources. Each ResourceClaim that Kubernetes generates from the template is bound to a specific Pod. When the Pod terminates, Kubernetes deletes the corresponding ResourceClaim.
- **DeviceClass:** Defines a category of devices that can be claimed and how to select specific device attributes in claims. The DeviceClass parameters can match zero or more devices in ResourceSlices. To claim devices from a DeviceClass, ResourceClaims select specific device attributes.
- **ResourceSlice:** Represents one or more resources that are attached to nodes, such as devices. Drivers create and manage ResourceSlices in the cluster.

Find more: <https://kubernetes.io/docs/concepts/scheduling-eviction/dynamic-resource-allocation/>

The driver publishes:

```
apiVersion: resource.k8s.io/v1beta1
kind: ResourceSlice
metadata:
---
spec:
  devices:
  - basic:
      attributes:
        architecture:
          string: Hopper
        brand:
          string: Nvidia
        cudaComputeCapability:
          version: 9.0.0
        cudaDriverVersion:
          version: 12.8.0
        driverVersion:
          version: 570.124.6
        index:
          int: 6
        minor:
          int: 6
        productName:
          string: NVIDIA H100 80GB HBM3
        capacity:
          memory:
            value: 81559Mi
```

The Cluster Administrator defines:

```
apiVersion: resource.k8s.io/v1beta1
kind: DeviceClass
metadata:
  name: gpu.nvidia.com
spec:
  selectors:
  - cel:
      expression: device.driver == 'gpu.nvidia.com' && device.attributes['gpu.nvidia.com'].type == 'gpu'
```

The Users create:

```
apiVersion: resource.k8s.io/v1beta1
kind: ResourceClaimTemplate
metadata:
  name: timesliced-gpu
spec:
  spec:
    devices:
      config:
        - opaque:
            driver: gpu.nvidia.com
            parameters:
              kind: GpuConfig
            apiVersion:
              resource.nvidia.com/v1beta1
            sharing:
              strategy: TimeSlicing
              timeSlicingConfig:
                interval: MEDIUM
            requests:
              - timesliced-gpu
```

```
apiVersion: v1
kind: Pod
metadata:
---
spec:
  containers:
  - name: ctr1
    ---
    resources:
      claims:
        - name: timesliced-gpu
          request: ts-gpu
        - name: ctr2
    ---
    resources:
      claims:
        - name: timesliced-gpu
          request: ts-gpu
      resourceClaims:
        - name: timesliced-gpu
          resourceClaimTemplateName: timesliced-gpu
```

The driver publishes:

```
apiVersion: resource.k8s.io/v1beta1
kind: ResourceSlice
metadata:
---
spec:
  devices:
  - basic:
      attributes:
        architecture:
          string: Hopper
        brand:
          string: Nvidia
        cudaComputeCapability:
          version: 9.0.0
        cudaDriverVersion:
          version: 12.8.0
        driverVersion:
          version: 570.124.6
      index:
        int: 6
      minor:
        int: 6
      productName:
        string: NVIDIA H100 80GB HBM3
      capacity:
        memory:
          value: 81559Mi
```

The Cluster Administrator defines:

```
apiVersion: resource.k8s.io/v1beta1
kind: DeviceClass
metadata:
  name: gpu.nvidia.com
spec:
  selectors:
  - cel:
      expression: device.driver == 'gpu.nvidia.com' && device.attributes['gpu.nvidia.com'].type == 'gpu'
```

The Users create:

```
apiVersion: resource.k8s.io/v1beta1
kind: ResourceClaimTemplate
metadata:
  name: timesliced-gpu
spec:
  spec:
    devices:
      config:
        - opaque:
            driver: gpu.nvidia.com
            parameters:
              kind: GpuCompute
            apiVersion:
              resource.nvidia.com/v1beta1
            sharing:
              strategy: TimeSlicing
              timeSlicingConfig:
                interval: MEDIUM
            requests:
              - timesliced-gpu
```

This is called
user-mediated sharing

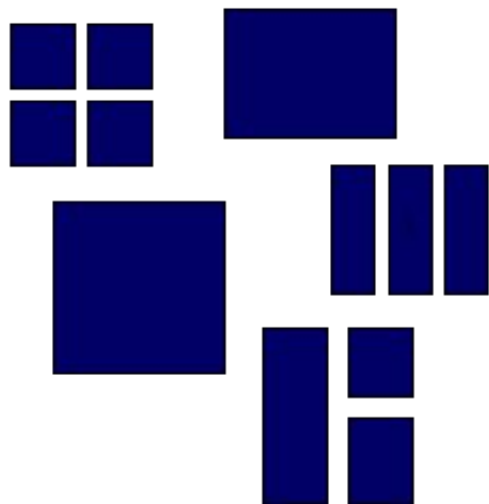
```
apiVersion: v1
kind: Pod
metadata:
  name: ctr1
spec:
  containers:
  - name: ctr1
    resources:
      claims:
      - name: timesliced-gpu
        request: ts-gpu
      name: ctr2
---
resources:
  claims:
  - name: timesliced-gpu
    request: ts-gpu
  resourceClaims:
  - name: timesliced-gpu
    resourceClaimTemplateName: timesliced-gpu
```

This is much more complicated than before
Why do we even want to use DRA?

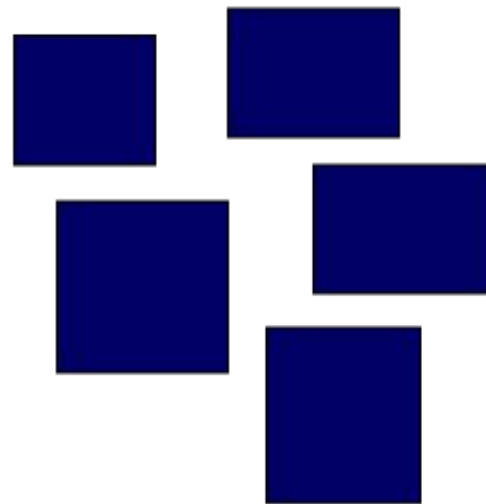
This is much more complicated than before
Why do we even want to use DRA?

Well, because it's much more powerful!

Before DRA:



With DRA



We don't need to pre-partition any of the GPUs!
No manual repartitioning required!
No per-node configuration!

Demo

Will be running workloads requiring

1. a full GPU
2. a time-sliced GPU between 2 and 8 process
3. a shared with MPS between 3 process

Without manually reconfiguring the GPU, the driver understand the request, talks with the scheduler and allocates accordingly.

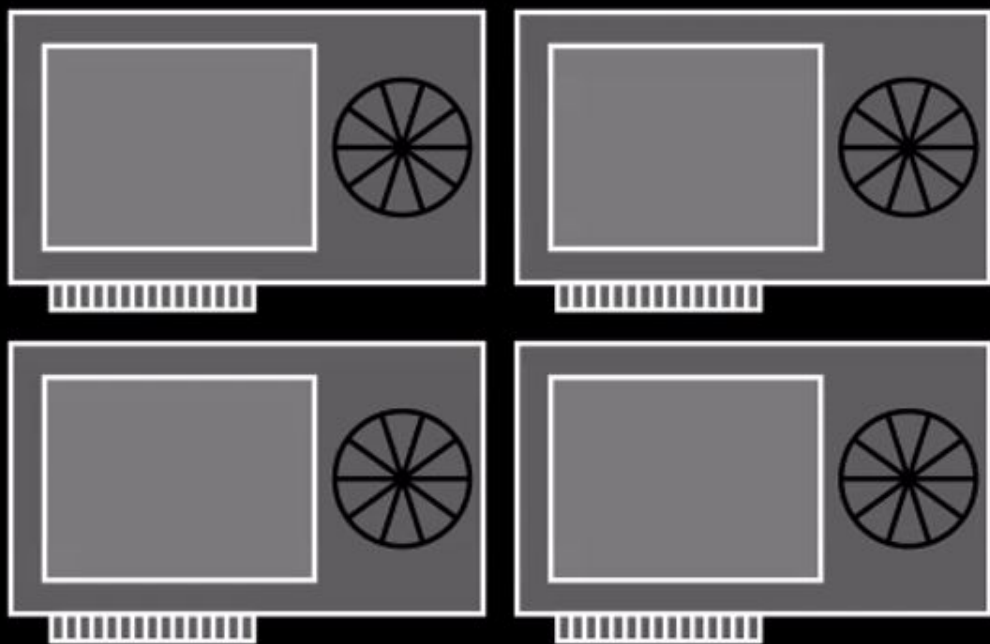
>

Every 0.1s: nvidia-smi kubecon-dra-001: Thu Mar 27 20:23:50 2025

Thu Mar 27 20:23:50 2025

```
+-----+
+-----+
| NVIDIA-SMI 570.124.06                Driver Version: 570.124.06
| CUDA Version: 12.8                  |
+-----+
+-----+
| GPU Name                               Persistence-M | Bus-Id        Disp.
+-----+-----+-----+
| A | Volatile Uncorr. ECC |
|   | Fan  Temp  Perf    Pwr:Usage/Cap |      Memory-Usag
e | GPU-Util  Compute M. |
|                                       |
|           MIG M. |
+=====+-----+
+-----+-----+-----+
|   0   NVIDIA A100-PCIE-40GB                Off |   00000000:00:06:0  Of
f |           0 |
| N/A   43C    P0               62W / 250W |     1MiB / 40960Mi
B |   0%      Default |
|                                       |
|           Disabled |
+-----+-----+-----+
+-----+
+-----+
+-----+
| Processes:
|
| GPU   GI    CI          PID    Type    Process name
|       ID    ID             |
|       ID    Usage         |
+=====+-----+
+=====+
| No running processes found
|
+-----+
+-----+
```

I



A couple of details to keep in mind

A couple of details to keep in mind

- Initially using DRA or the device plugin were mutual exclusive. K8s 1.36 makes it possible to support both thanks to [KEP-5004: DRA Extended Resource](#)

A couple of details to keep in mind

- In the first driver versions, Dynamic allocation of MIG devices was not yet available. Now it's been available since [v25.12.0](#)!

A couple of details to keep in mind

- In the first driver versions, Dynamic allocation of MIG devices was not yet available. Now it's been available since [v25.12.0](#)!
 - It also needs kubernetes 1.36, that brings [KEP-4815: DRA Partitionable Devices](#)
 - But it needs a Hopper GPU, or newer: *[“Starting with the Hopper generation of GPUs, enabling MIG mode no longer requires a GPU reset to take effect \(and thus the driver does not attempt to reset the GPU in the background\).”](#)*

A couple of details to keep in mind

- k8s-device-plugin only supports system-mediated sharing, while in the beginning, the DRA driver only supported user-mediated sharing.
 - This meant with DRA, GPUs couldn't be shared across namespaces.

A couple of details to keep in mind

- k8s-device-plugin only supports system-mediated sharing, while in the beginning, the DRA driver only supported user-mediated sharing.
 - This meant with DRA, GPUs couldn't be shared across namespaces.

“With added support for Partitionable Devices, DRA is capable to do both user-mediated sharing and system-mediated sharing. The system-mediated sharing support will be implemented by advertising multiple “replicas” of the same underlying GPU, much like the device plugin does. The nice thing, however, will be that every GPU will still be usable as an exclusive GPU until it is first allocated as a sharable one. With the device plugin, this is not possible because each GPU must either be advertised as either replicated or exclusive -- this decision cannot be made dynamically.”

Let's benchmark the new setup

- Kubernetes 1.32.0
- A100 40GB PCIe
- GPU access using DRA (NVIDIA k8s-dra-driver-gpu v25.3.0-rc.2)

Use Case 1: LQFT

Simulation of Quantum Chromodynamics

Atomic nuclei are held together by the strong force which can be described by the theory of Quantum Chromodynamics (QCD). Because of the probabilistic nature of quantum mechanics, you need to account for all the possible ways an interaction between particles could happen. As there are infinite possibilities for interactions between quarks and gluons, this becomes intractable. In order to solve this problem, space and time are discretized into a finite four dimensional grid (three dimensions for space and one dimension for time). In this grid each point represents the quark field, and the lines connecting them represent the gluon field. Calculations in lattice QCD can be done on GPUs, leveraging NVIDIA's CUDA platform.

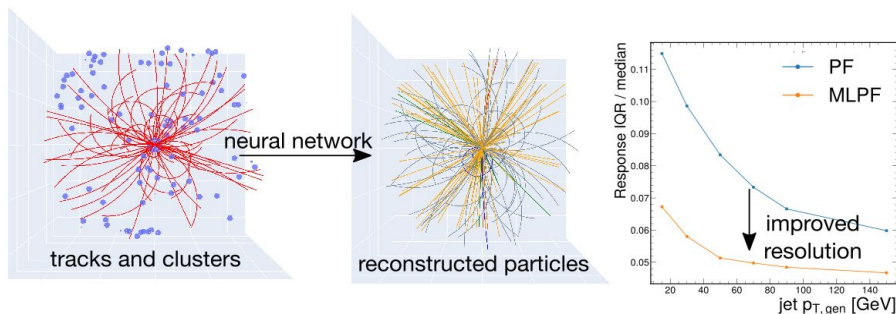
QUDA is a library for performing calculations in lattice QCD on GPUs. It leverages NVIDIA's CUDA platform to solve linear systems in order to simulate the QCD interactions.

Find more: <https://github.com/lattice/quda>

Use Case 2: MLPF

Event Reconstruction

When particles collide in the LHC they interact with each other and new particles get created. In High Energy Physics, this is typically referred to as an event. It is constituted of interacting particles, e.g. colliding protons, an interaction point, e.g. the center of the ATLAS or CMS detector, and outgoing particles, e.g. a Higgs boson that got produced in the process. An event is then usually recorded using particle detectors and the tracked particles form the so-called event signature. This signature is unique for the event and contains the produced particles, their type, and several properties like energies or momentum. One computational task is the identification and reconstruction of the comprehensive list of produced particles. These can be photons, electrons, muons, or charged and neutral hadrons.



MLPF

Machine-learned particle flow reconstruction (MLPF) is a project which implements full event reconstruction using state-of-the-art machine learning models. The input data consists of particle tracks and energy data from the calorimeters. It outputs the identity of the particles as well as their physics properties like momentum and energy.

Version used: v.1.6.2 (preliminary code)

> Pata, J., Wulff, E., Mokhtar, F. et al. Improved particle-flow event reconstruction with scalable neural networks for current and future particle detectors. Commun Phys 7, 124 (2024).

<https://doi.org/10.1038/s42005-024-01599-5>

Find more: <https://github.com/jpata/particleflow>

Benchmarking on a full GPU

LQFT:

Input	Execution Time
24x24x24x24	32.671s
32x32x32x32	98.314s
48x48x32x32	219.129s
48x48x48x32	331.617s

MLPF:

Input	Time
10000	72s
25000	110s
50000	169s
100000	286s
250000	640s
500000	1224s

2 Processes running on the same GPU with Time-slicing

LQFT:

Input	2 Processes on a full GPU - Sequentially	2 Processes - Time-slicing
24x24x24x24	65.342s	74.797s
32x32x32x32	196.628s	198.657s
48x48x32x32	438.258	434.053s
48x48x48x32	663.234s	636.196s

Because time-slicing gives you access to the full GPU compute power, some programs run as fast as if they were run sequentially, plus some context-switching overhead.

MLPF:

Input	2 Processes on a full GPU - Sequentially	2 Processes - Time-slicing
10000	144s	287s
25000	220s	651s
50000	338s	1248s

Of course for some programs, the context-switching overhead is non-negligible.

4 Processes running on the same GPU with Time-slicing

LQFT:

Input	4 Processes on a full GPU - Sequentially	4 Processes - Time-slicing
24x24x24x24	130.684s	152.798s
32x32x32x32	393.256s	389.491s
48x48x32x32	876.516s	OOM
48x48x48x32	1326.468s	OOM



With time-slicing we should be aware that all the processes will require some additional memory to hold the state between context switching.

MLPF:

Input	4 Processes on a full GPU - Sequentially	4 Processes - Time-slicing
10000	288s	634s
25000	440s	1446s
50000	676s	2121s

2 Processes running on the same GPU with MPS

LQFT:

Input	2 Processes on a full GPU - Sequentially	2 Processes - MPS
24x24x24x24	65.342s	59.072s
32x32x32x32	196.628s	183.364s
48x48x32x32	438.258	413.989s
48x48x48x32	663.234s	626.382s

Sharing with MPS results in excellent performance since we have enough resources to efficiently run 2 processes in parallel on the same GPU.

MLPF:

Input	2 Processes on a full GPU - Sequentially	2 Processes - MPS
10000	144s	83s
25000	220s	132s
50000	338s	210s
100000	572s	364s

4 Processes running on the same GPU with MPS

LQFT:

Input	4 Processes on a full GPU - Sequentially	4 Processes - MPS
24x24x24x24	130.684s	181.909s
32x32x32x32	393.256s	569.743s
48x48x32x32	876.516s	1289.93s
48x48x48x32	1326.468s	OOM



With 4 processes we see a drop in performance likely due to memory bandwidth saturation, cache thrashing, and managing long-running kernels concurrently.

MLPF:

Input	4 Processes on a full GPU - Sequentially	4 Processes - MPS
10000	288s	102s
25000	440s	174s
50000	676s	296s
100000	1144s	525s

Conclusions

1. The performance of time-slicing and MPS depends heavily on the workload profile.
2. MPS works great when workloads underutilize the GPU because the idle resources can be used to run other processes in parallel.
3. Time-slicing makes sense for bigger workloads since it avoids resource contention (3 x 80% GPU util works on time-slicing, but not on MPS).
4. Time-slicing can severely underperform when the process state is complex, increasing the context-switching times.

Running LQFT on a MIG partitioned GPU

Input	1g.5gb	2g.10gb	4g.20gb
24x24x24x24	169.610s	161.076s	57.353s
32x32x32x32	531.022s	507.139s	179.360s
48x48x32x32	OOM	1141.458s	401.431s
48x48x48x32	OOM	OOM	603.042s

Input	1g.5gb / 2g.10gb	2g.10gb / 4g.20gb
24x24x24x24	1.05	2.80
32x32x32x32	1.04	2.82
48x48x32x32	-	2.84
48x48x48x32	-	-

Running MLPF on a MIG partitioned GPU

Input	Dedicated GP	1g.5gb	2g.10gb	4g.20gb
10000	72s	OOM	104s	86s
25000	110s	OOM	182s	134s
50000	169s	OOM	300s	212s

Input	1g.5gb / 2g.10gb	2g.10gb / 4g.20gb
10000	-	1.20
25000	-	1.35
50000	-	1.41

MIG Conclusions

1. The smallest MIG partition can be too small for many use cases.
2. Request reasonably sized MIG partitions.
3. MIG isolates the compute and memory between workloads. Since it behaves as smaller individual GPUs, we don't need much information about the use cases.

Monitoring GPUs

NVIDIA Data Center GPU Manager (DCGM) - a powerful tool designed to manage and monitor NVIDIA GPUs in data centers, ensuring optimal performance and reliability. It helps track:

- the health
- performance
- utilization

You can either use the command line tool `dcgmi`, or DCGM Exporter to export the metrics.

Use field identifiers to enable customize the metrics reported:

<https://docs.nvidia.com/datacenter/dcg/2.4/dcg-api/dcg-api-field-ids.html>

Find more: <https://developer.nvidia.com/dcg>

Final Conclusions

1. Using GPUs by leveraging DRA is not that complicated once you get used to the API.
2. The DRA approach is still in active development. **Try it out, get involved and provide feedback!**
3. Different sharing mechanisms work best on different use cases, understand the implications and choose accordingly.
4. Consider using the full GPU for use cases that will make use of them.
5. Always ensure you have detailed GPU monitoring in your infrastructure.

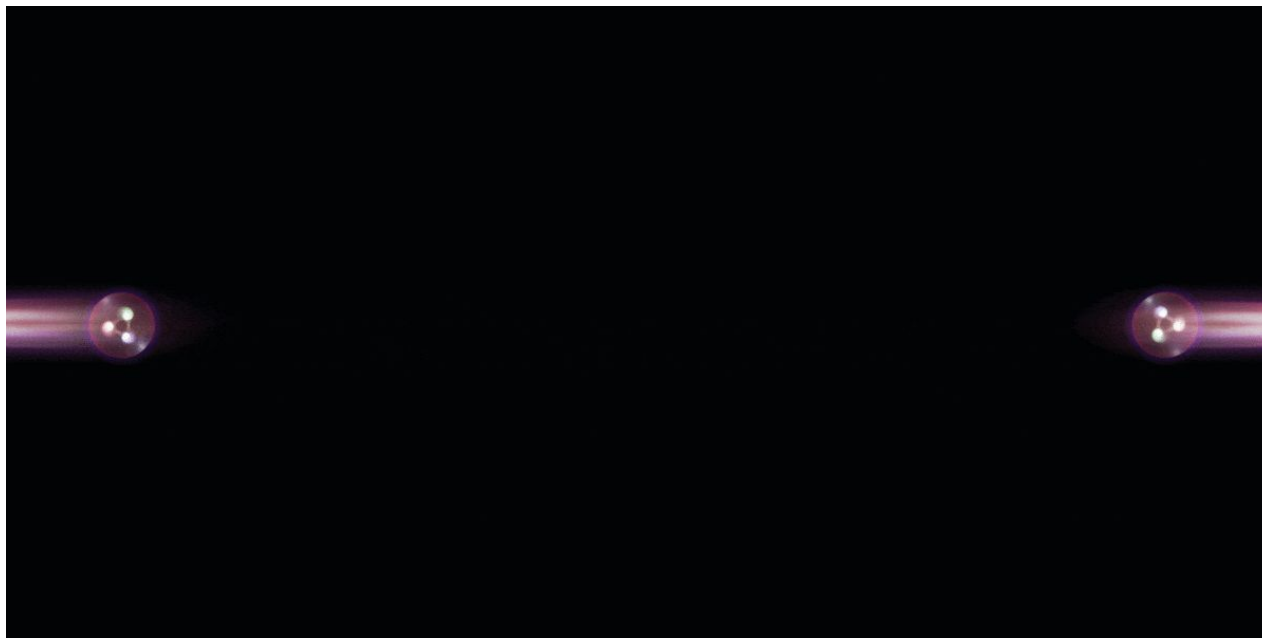
Final Final Conclusions

If you would like to try this out but don't have a gpu available, you could try <https://github.com/kubernetes-sigs/dra-example-driver>

This is a intended to be a starting point when building a driver.

The driver itself provides access to **a set of mock GPU devices**, and this demo walks through the process of building and installing the driver followed by running a set of workloads that consume these GPUs.

Thank You!



This is a representation of a collision between two protons, made up of three quarks each. Such collisions happen hundred of millions of times per second in each of the Large Hadron Collider detectors.